Kotlin
Cheat Sheet

by ekito
agiuliani@ekito.fr
@arnogiu

Kotlin is an open source statically typed language for the JVM. It can run on Java 6+ and bring smart features to make your code concise and safe. Its high interoperability helps to adopt it very quickly. Official documentation can be found at http://kotlinlang.org/

GETTING STARTED

Gradle

```
buildscript {
    ext.kotlin_version = '<version to use>'

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

apply plugin: "kotlin"
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

Android

In your build.gradle, use gradle setup and the android-kotlin plugins:

```
android {
    sourceSets {
        main.java.srcDirs += 'src/main/kotlin'
    }
}

apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions' // if use extensions
```

Gradle Options

Compilation tuning in your gradle.properties file:

```
# Kotlin
kotlin.incremental=true
# Android Studio 2.2+
android.enableBuildCache=true
```

Maven

Refer to the documentation online : http://kotlinlang.org/docs/reference/using-maven.html

BASICS

```
package mypackage

import com.package

/** A block comment
 */
// line comment
```

by ekito
agiuliani@ekito.fr
@arnogiu

Values & Variables Definition

Val represents a constant value & **var** a mutable value

```
val a: Int = 1
val b = 1    // `Int` type is inferred
val c: Int  // Type required when no initializer is provided
c = 1        // definite assignment
```

```
var x = 5 // `Int` type is inferred
x += 1
```

NULL SAFETY & OPTIONAL TYPING

Optional typing with <TYPE>**?**

```
var stringA: String = "foo"
stringA = null //Compilation error - stringA is a String (non optional) and can't have null value
var stringB: String? = "bar" // stringB is an Optional String
stringB = null //ok
```

Safe call (**?.**) or explicit call (**!!.**)

```
val length = stringB.length // Compilation error - stringB can be null !
val length = stringB?.length //Value or null - length is type Int?
val length = stringB!!.length // Value or explicit throw NullPointerException - length is type Int
```

Defining default value with elvis operator (**?:**)

```
// set length default value manually
val length = if (stringB != null) stringB.length else -1
//or with the Elvis operator
val length = stringB?.length ?: -1
```

Late variable initialization with **lateinit**. You can also look at **lazy** initialized values

```
lateinit var myString : String // lets you define a value later, but is considered as null if not set
val myValue : String by lazy { "your value ..." }
```

CLASSES

A simple Java POJO (Getter, Setter, Constructor) in one line

```
class User (
        var firstName: String,
        var lastName:  String,
        var address:  String? = null
)
```

Public Visibility by default

All is public by default. Available visibility modifiers are: **private, protected, internal, public**

Data Class

By adding **data** keyword to your class, add toString(), equals(), copy() and exploded data (see destructuring data below) capabilities

Kotlin
Cheat Sheet

by ekito
agiuliani@ekito.fr
@arnogiu

Properties
Properties can be declared in constructor or class body. You can also limit access to read (get) or write (set) to any property.

```
class User() {//primary empty constructor
    constructor(fn: String) : this() { //secondary constructor must call first one
        firstName = fn
    }
    var firstName: String = ""
    val isFilled: Boolean // read only access property
        get() = !firstName.isEmpty()
}
```

No Static, use *Object* !
You can write singleton class, or write companion class methods:

```
// my resource singleton
object Resource {
    // properties, functions …
}
```

Closed Inheritance
The **:** operator, makes inheritance between classes and is allowed by opening it with **open** modifier

```
open class A()
class B() :A()
```

FUNCTIONS
Function can be defined in a class (aka method) or directly in a package. Functions are declared using the **fun** keyword

Default Values
Each parameter can have a default value

```
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size()) {
    //...
}
```

Named Arguments
When calling a function, you can freely set the given parameters by its order or by its name:

```
read(myBytes, 0, myBytes.length) // old way to call
reformat(myBytes, len = 128) // using default values & named params
```

Function Extension
Kotlin allows you to define a function to add to an existing Class

```
fun String.hello(): String = "Hello " + this
// use the new hello() method
val hi = "Kotlin !".hello()
```

by ekito
agiuliani@ekito.fr
@arnogiu

Lambda
A lambda expression or an anonymous function is a “function literal”, i.e. a function that is not declared, but passed immediately as an expression

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
```

- A lambda expression is always surrounded by curly braces,
- Its parameters (if any) are declared before -> (parameter types may be omitted),
- The body goes after -> (when present).

Destructuring Data
Sometimes it is convenient to destructure an object into a number of variables. Here is the easy way to return two values from a function:

```
fun extractDelimiter(input: String): Pair<String, String> = …
val (separator, numberString) = extractDelimiter(input)
```

Data classes can be accessed with destructured declaration.

WHEN - A better flow control
when replaces the old C-like switch operator:

```
when (s) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // Note the block
        print("x is neither 1 nor 2")
    }
}
```

It can also be used for pattern matching, with expressions, ranges and operators (is, as …)

COLLECTIONS
Collections are the same as the ones that come with Java. Be aware that Kotlin makes difference between immutable collections and mutables ones. Collection interfaces are immutable.

```
// immutable list
val list = listOf("a", "b", "c","aa")
list.filter { it.startsWith("a") }
```

```
// map loop with direct access to key and value
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
for ((k, v) in map) {
    println("$k -> $v")
}
```

Maps and Arrays can be accessed directly with [] syntax or with range expressions. Various of methods on list/map can be used with lambdas expression :

```
// write with mutable map
map["a"] = "my value"
// filter collection
items.filter { it % 2 == 0 }
```